REINFORCEMENT LEARNING FOR AUTONOMOUS VEHICLES

An Undergraduate Research Scholars Thesis

by

AMOGH PANDEY

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dileep Kalathil

May 2021

Major: Electrical Engineering

TABLE OF CONTENTS

ABSTRACT

Reinforcement Learning for Autonomous Vehicles

Amogh Pandey
Department of Electrical and Computer Engineering
Texas A&M University

Research Faculty Advisor: Dileep Kalathil
Department of Electrical and Computer Engineering
Texas A&M University

In this project, we implement and deploy reinforcement learning (RL) algorithms for path planning, decision making, and navigation tasks in autonomous vehicles, and compare them to traditional control algorithms. Various control and reinforcement learning algorithms are deployed in a simulation environment to test their performance on various navigation tasks, and will finally be deployed on a robotic car to study the performance of RL in real life autonomous driving tasks. Along the way, the challenges of mapping and localization for the physical car are explored. Furthermore, our results and observations will be used to hopefully establish RL as a viable alternative to control theory for autonomous navigation related tasks.

ACKNOWLEDGMENTS

I would like to thank my faculty advisor, Dr. Dileep Kalathil, for his guidance and support throught the course of this project. Under his guidance, I have been able to discover my passion for research, and have learned a lot about driving a research project from its initial stages to near completion. I would also like to thank Dr. Srinivas Shakkottai for his guidance and direction on this project. Finally, I would like to thank my graduate student mentors Akshay Sarvesh, Desik Rengarajan, Gargi Vaidya, and Vishnu Saj for their help and guidance throughout the duration of this project. Getting into research as an undergraduate student initially seemed to be very daunting, but with their help, support, and guidance, I never felt alone in the process.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

1. INTRODUCTION

Reinforcement learning (RL) is a branch of machine learning that typically deals with solving decision making and path planning problems. Such problems fall under the general mathematical framework known as Markov Decision Processes (MDP). An MDP consists of a set of states S, set of actions A, a reward function $R(s, a, s')$, and a set of transition probabilities $P(s'|s, a)$, where s' is the next state. The solution to an MDP is known as a policy $\pi(a|s)$, which is a mapping from the set of states S to the set of actions A, for all states $s \in S$. The optimal solution to an MDP, known as the optimal policy $\pi^*(a|s)$, is the policy that maximizes the notion of an expected sum of rewards that an agent in the environment would receive.

Traditionally, path planning algorithms(such as A*) and optimal control theory have been used to solve navigation tasks. Proportional-Integral-Derivative (PID) Control [1] and Model Predictive Control (MPC) [2] [3] are examples of some commonly used control techniques for navigation tasks. The inherent problem with control algorithms is the need for an accurate model of the system, and stability and robustness in a stochastic world, which includes challenges adapting to new circumstances/environments. RL addresses these problem by presenting model-free approaches to generate optimal policies, and by "learning" in a simulation environment, RL is able to easily adapt to the challenges presented by a stochastic world.

In this project, we explore RL for autonomous navigation and driving. We break down autonomous driving into two parts, path planning and control, and explore these tasks in simulation before moving to reality. To explore RL in reality, we have a 1/18 scale robotic car made by Amazon Web Services (AWS), called the AWS DeepRacer [4], which is equipped with stereo cameras, a 360° LIDAR sensor (12 meter range), and an onboard Ubuntu machine running Robot Operating System (ROS). We hope to establish RL as a viable alternative to traditional control methods for autonomous navigation related tasks.

2. SUBSYSTEMS

The goal of this project is to explore Reinforcement Learning for autonomous vehicles, and to try to establish RL as a viable alternative to traditional/optimal control methods. Therefore, this project is divided into three main subsystems: Control/RL Algorithms, Simulations, and Physical Robotic Car Implementation.

Control/RL Algorithm: Control algorithms will be implemented to dictate a control policy for the autonomous car to follow to accomplish a certain task. Then, RL algorithms will be implemented to be capable of effectively training a model to complete the given task or set of tasks. This subsystem also consists of Path Planning algorithms, will will be used to generate a path for the vehicle to follow from start to goal.

Simulator: The simulator allows us to have a practical way of gauging the performance of control algorithms. In the case of RL algorithms, the simulator facilitates training and allows us to generate a trained model (optimal policy) which can then be deployed in another instance of a simulated vehicle and its performance can be evaluated.

Physical AWS DeepRacer: After tuned control policies or trained RL models have been generated and evaluated in simulation, they will be deployed on the DeepRacer car and their performance in a real-life setting/environment will be evaluated. However, prior to the deployment on any algorithms on the physical car, the challenges of mapping and localization are addressed.

Figure 2.1: Project Breakdown.

2.1 Path Planning Algorithms

The algorithms subsystem relates to the actual design and implementation of the path planning algorithms used to perform autonomous navigation, along with the control and RL algorithms. Autonomous navigation can be broken down into two main tasks: Path Planning and Control/Actuation (explored in later sections). This is because in an autonomous navigation task, typically the goal is to get from point A to point B, in an environment that contains obstacles. In order to guide a robot from A to B, we need a path. Path planning algorithms are used to generate a set of waypoints along an optimal path from A to B. Next, once we have the waypoints, the robot needs to be able to move itself from one waypoint to the next, until it reaches the goal. Somehow, the robot must be able to take as an input the set of waypoints, and actuate its throttle/steering angle to correctly guide itself along the path given by the waypoints. This is not a trivial task and consequently requires the use of control algorithms (such as PID control) or RL.

The first step was to work on path planning algorithms. The A* search algorithm is a very common algorithm for path planning, and is guaranteed to give an optimal solution (so long as the heuristic function used is admissible), so this is what was implemented first. An off- the-shelf A* algorithm was used, as A* has been around for many decades now, and people have already coded it very efficiently. For our navigation path planning purpose, the cost function was the L1 distance (Manhattan Distance) between the start point and end point, and the heuristic function was the L2 distance (Euclidean distance) between the start point and end point. The cost and heuristic function were chosen such that the heuristic was admissible (meaning that A* would find the optimal path). In order for the heuristic to be admissible, it must always be an underestimate of the true cost. Clearly, for any two points A and B, the L2 norm is always less than or equal to the L1 norm.

A map of a typical 4 room floor plan with obstacles was implemented as a matrix, and fed to the A* algorithm, which output an optimal path **Figure 2.2**.

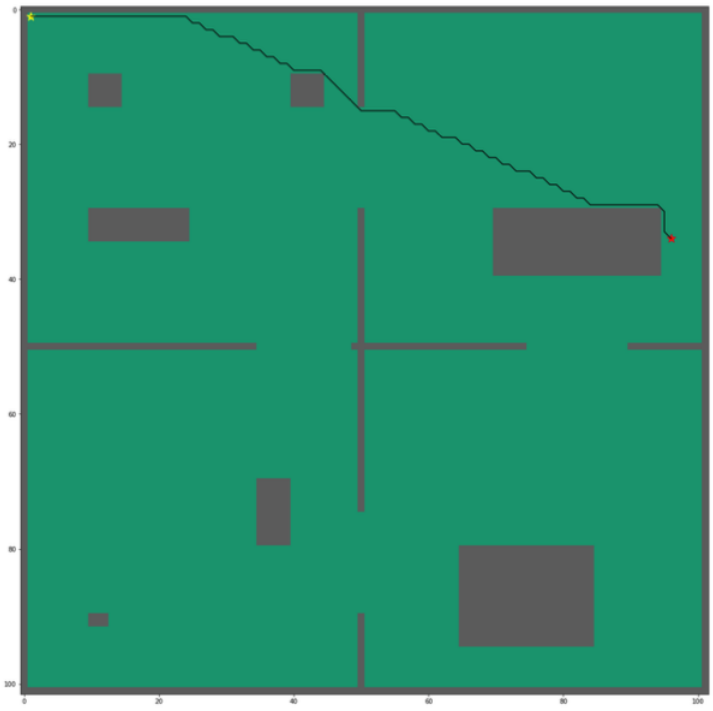

Figure 2.2: Off-the-Shelf A* Path in Four Room Environment.

While the off-the-shelf A* algorithm performed well in obtaining the optimal path, the path it found is quite impractical for autonomous cars or drones. The path is very close to walls and obstacles. While this may indeed be an optimal path, it is an unsafe path in a real life (stochastic) setting, as a small accident could push a car into an obstacle/wall, or a gust of wind could blow a drone into an obstacle/wall causing its propellers to break and it to crash.

To remedy this, we had to create a modified "safe" A* algorithm that could still find an optimal path while considering safety as well. The working principle is this modification is to introduce an additional cost, called the n_score, which captures information about the proximity of obstacles within an x by x square of each position. Objects closer to the current position contribute more to the n_score than those further away in that x by x square. This n_score is added to the regular distance cost of each grid position, and the A* algorithm does its search based on this new cost function. In essence, since the search algorithm is trying to find the optimal (least cost) path,

it will find the optimal path to the goal position while keeping away from obstacles (since going close to obstacles results in a higher cost due to the n_score). The results of the modified "safe" A* algorithm are shown in **Figure 2.3**. Clearly, this path is keeping away from obstacles and walls, and is consequently a safer path than the one generated by the off-the-shelf A* algorithm. A second test case is shown **Figure 2.4**.

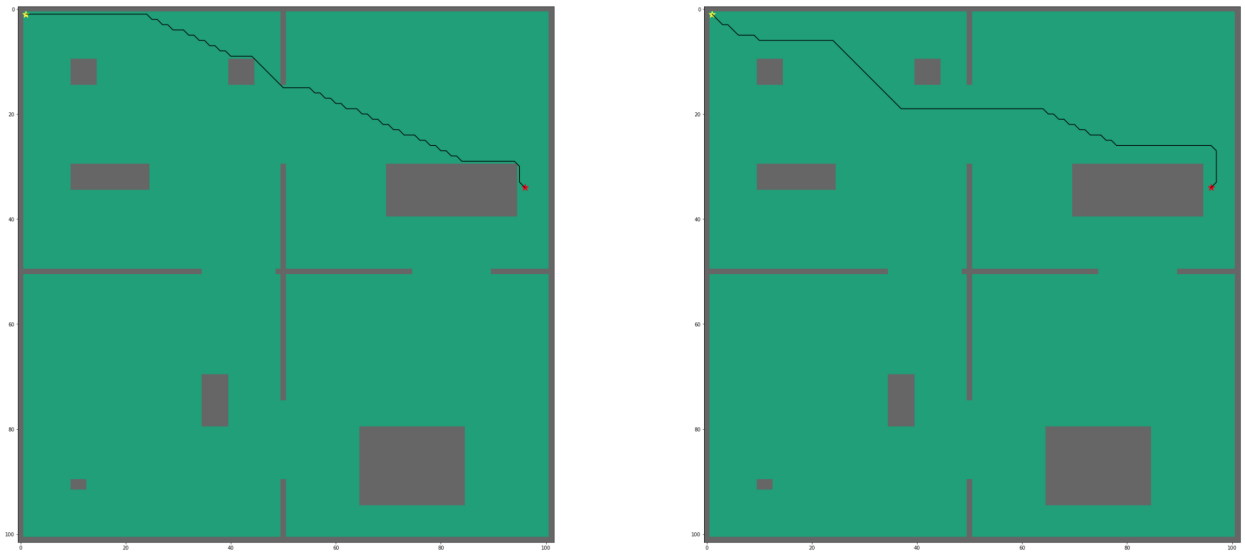

Figure 2.3: A* and Modified A* in Four Room Environment-Test Case 1.

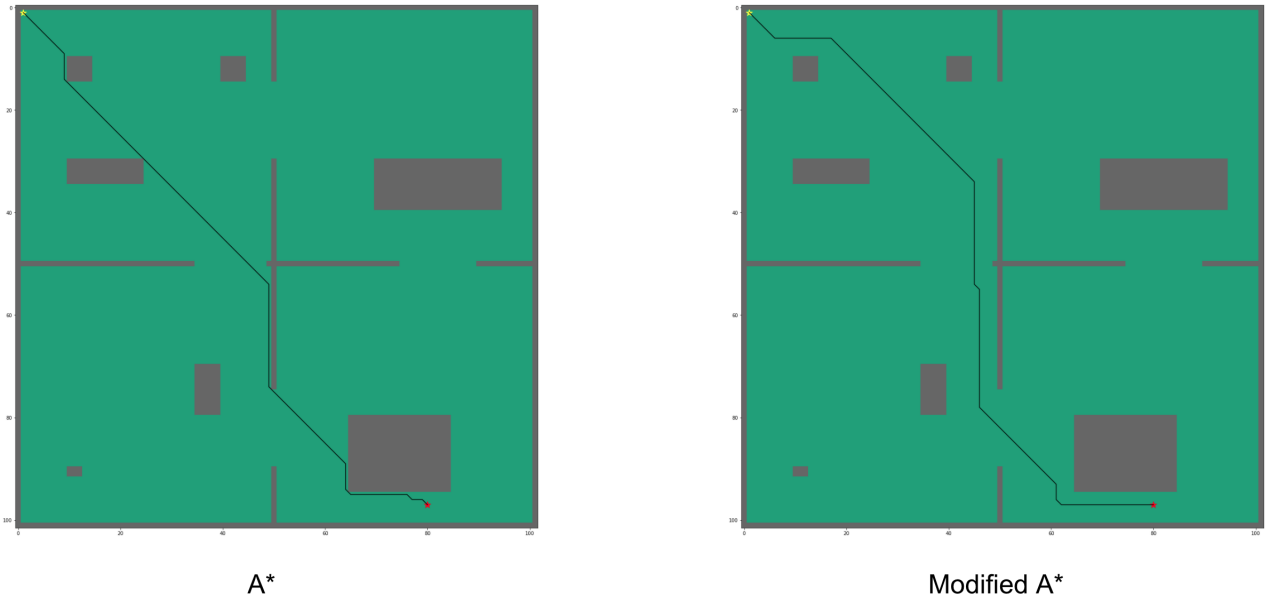

Figure 2.4: A* and Modified A* in Four Room Environment-Test Case 2.

However, the modified "safe" A* path, while better than the off-the-shelf path, still is not a practical path because it is very "rough" or "jagged", as it consists of many sharp turns (going down, then right, then down, then right etc.). It is not realistic to expect a car or drone to make such jerky movements to follow the jagged path. A smoother path would be more practical.

To obtain a smoother path, we made use of Savitzky-Golay filters. Simply put, path smoothing is essentially a low-pass filtering problem. In a low pass filter, we attenuate high frequency (quickly changing) components. Similarly, in path smoothing, we want to retain the original path, but smoothen out the frequent variations/changes in the path. Savitzky-Golay filters essentially do local low-pass filtering in a given window. After filtering (smoothing) a window (set of n waypoints), the filter shifts right by one, and again does the filters, and repeats this process this the end of the signal (set of waypoints). The resulting path, after passing the modified "safe" path through the Savitzky-Golay filter with a window size of 11 is shown in **Figure 2.5** on the right. Clearly the resulting path is a smoothed version of the modified "safe" path shown in **Figure 2.5** on the left.

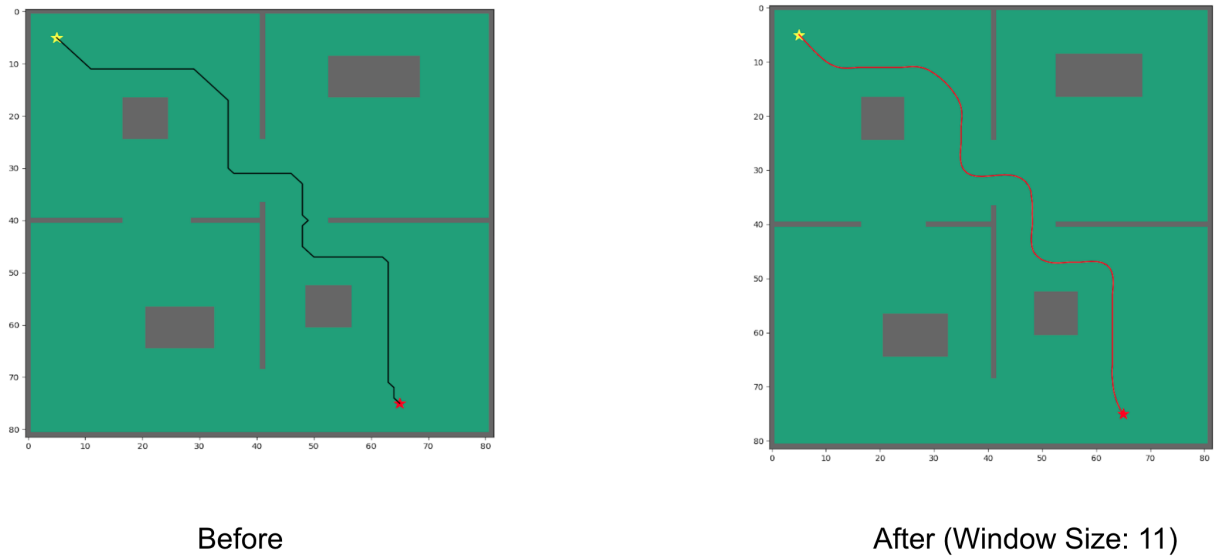

Figure 2.5: Before and After Path Smoothing.

We explored the effects of window size on the smoothened path, and found the following results, shown in **Figure 2.6**. For intuition, consider taking the average of 11 numbers (window size 11), versus taking the average of 41 numbers (window size 41). In the set of 11 numbers, if one number were to drastically change, the average will be affected a lot more than if one number in the set of 41 numbers were to drastically change. In the context of path smoothing, by considering a larger window size (which encompasses a larger number of waypoints that make up the path), we will end up with a much smoother path than simply having a smaller window size. Clearly, larger window sizes result in smoother paths, as expected.

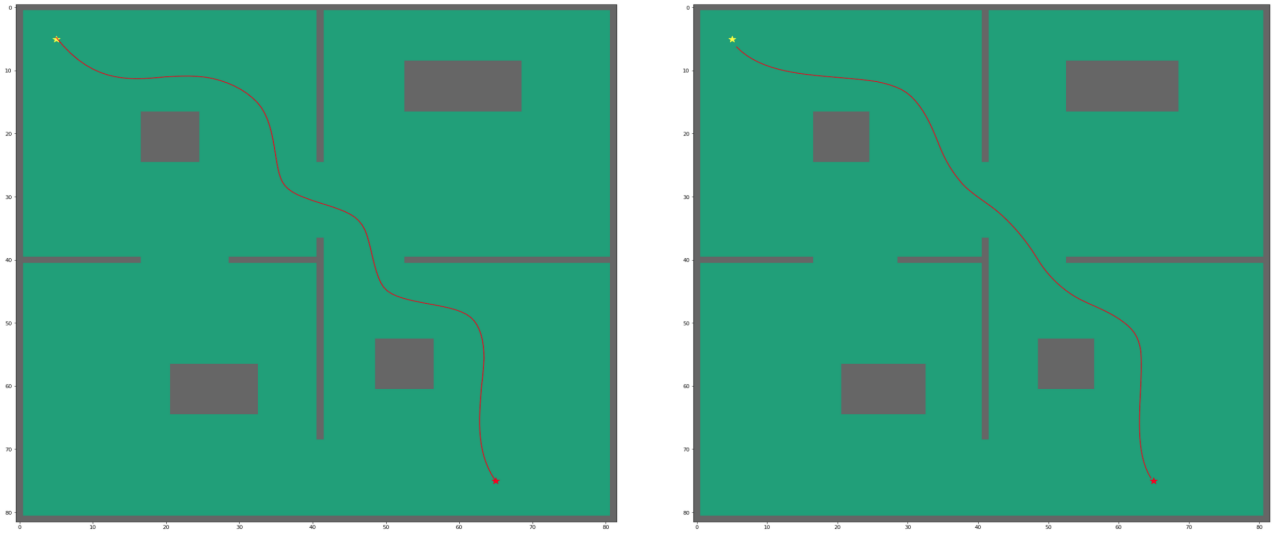

Figure 2.6: Varying Window Sizes.

2.2 Path Planning and PID Control Simulations

The simulator subsystem deals with the implementation of path planning, control, and RL algorithms in a virtual simulation environment. The simulator being used is Gazebo, which is a virtual physics environment commonly used for robotics simulations. Quite a bit of time was spent on getting this Gazebo environment installed on Linux machines and ensuring that all packages and dependencies that are needed for running simulations are in place and working properly, and getting the environment up and running. Then, we had to import over models of our vehicles (car and drone) into the Gazebo simulator so that we could visualize the algorithms on an agent. Finally, integrating the python algorithms with the Gazebo simulator also took some trial and error. After the simulator was set up, we had to design the environments in Gazebo. These environments had to match the environments that were fed to the A* algorithms in the previous step as matrices with 0 and 1 as elements. We then were able to start testing out the control algorithms. This process of simulation has been going on for a while, and will continue throughout the duration of this project, because as long as algorithms are being coded, they will need to be simulated and visualized so they can eventually be deployed in the physical robot car.

The first simulations we did were to test an off-the-shelf standard A* algorithm on an agent in a simple Gazebo environment. The purpose of this test was to ensure that the Gazebo environment was set up correctly and that the off-the-shelf A* algorithm we found was indeed working correctly (A* should always find the optimal path). **Figures 2.7 and 2.8** below demonstrate the two tests we did. The agent would start in the top-right corner and had the task of navigating to the square containing the gold star. The highlighted paths show some possible paths the agent could have taken from the start to the destination, and the green path showed the optimal path. In both cases, the agent followed the optimal path to the destination.

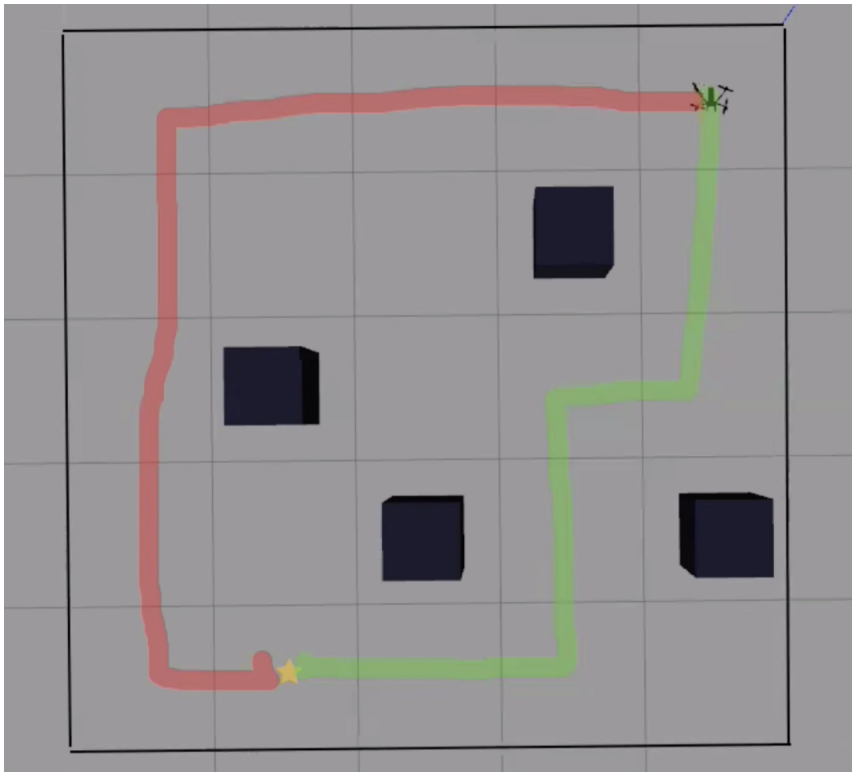

Figure 2.7: Off-the-shelf A* test 1.

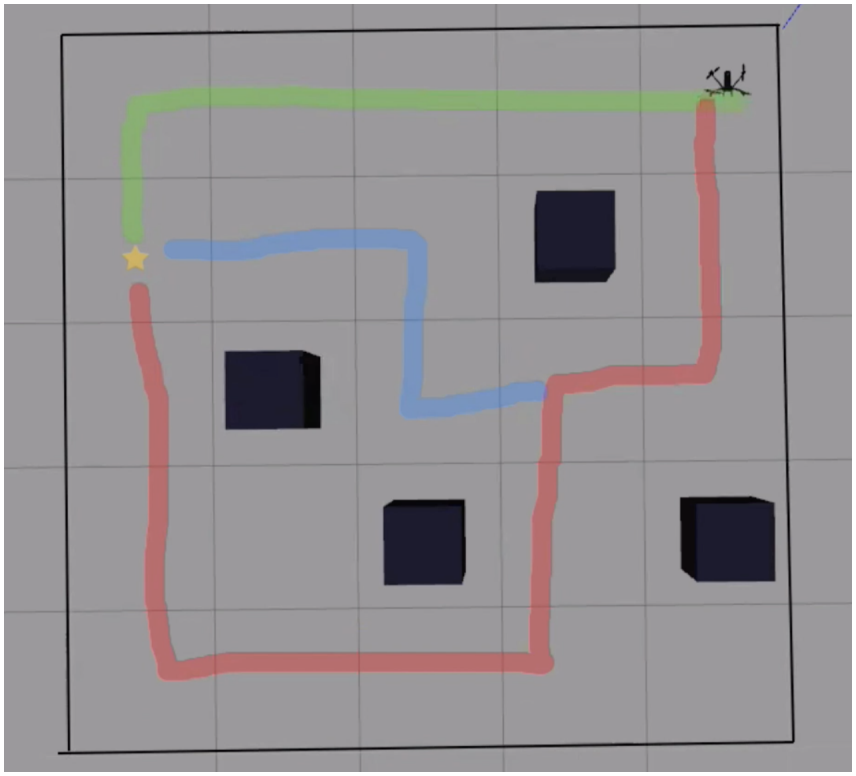

Figure 2.8: Off-the-shelf A* test 2.

After ensuring that the Gazebo environment is working properly, we first replicated the environment shown in **Figures 2.5 and 2.6** in Gazebo. Then we designed a PID controller capable of tracking the waypoints output by the modified "safe" A* algorithm (**Figures 2.5 and 2.6**). A screenshot of the agent following the modified "safe" A* path using the PID controller is shown in **Figure 2.9**.

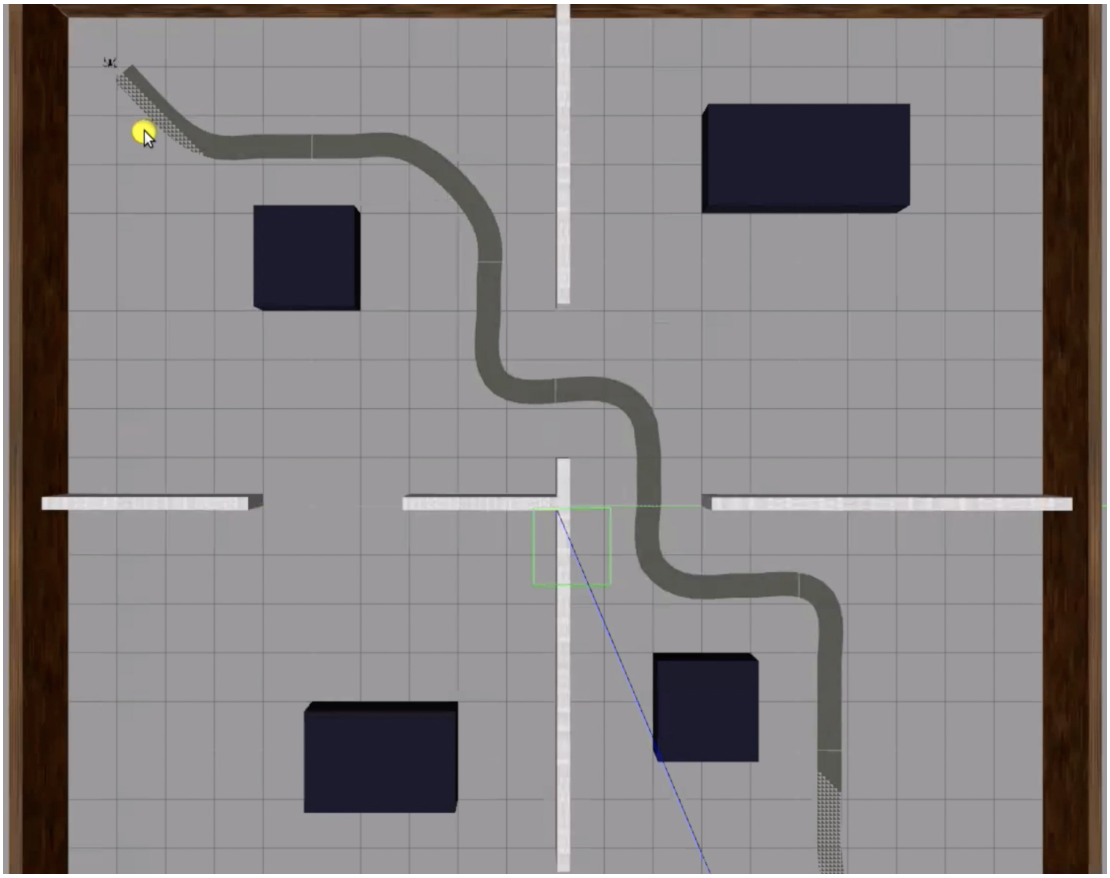

Figure 2.9: Modified "Safe" A* and PID Controller.

2.3 Physical Robotic Car Implementation

Ultimately, the goal of creating/implementing all of these algorithms is to be able to deploy them into the real world. The algorithms first go into the simulator so they can be assessed in a virtual, safe environment, where if anything goes wrong, it can be shut down with the click of a button. After the algorithms have performed well in the simulator, they can be moved to the physical robotic car. The car we are using is the AWS DeepRacer **Figure 2.10**. The car is equipped with two cameras and a LIDAR sensor.

Figure 2.10: AWS DeepRacer.

One problem that was found on the car related to the concept of mapping and localization. In the simulator, the position (x,y,z) coordinates with respect to the origin, along with other inertial measurements of the car/drone are known at each and every time instant, because the simulator is capable of making these measurements. The measurements can be used to navigate and control the vehicle. However, in reality, the physical robot is not able to provide its current coordinates because it is not equipped with a GPS. Knowledge of the real-time position of the robot is crucial because it needs to know where it is relative to the goal so it can keep on the given path. To solve this problem, we first tried integrating an IMU (Inertial Measurement Unit) onto the car.

The IMU we used is called the Razor 9DOF IMU MPU 9250 made by SparkFun. It provides us with information such as Roll, Pitch, Yaw, Linear Acceleration and Angular Velocity (in x, y, and z directions). Visualization of the IMU data is shown in **Figure 2.11**.

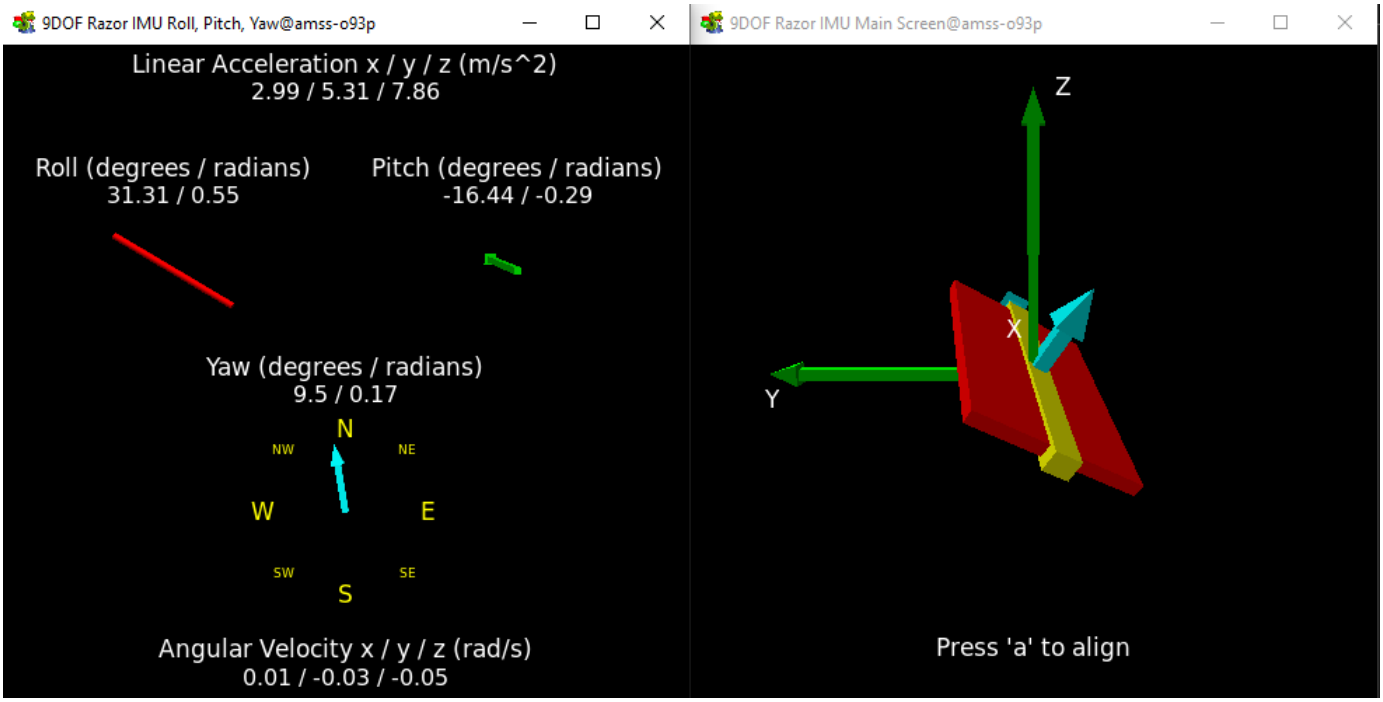

Figure 2.11: IMU Data Visualization.

The raw IMU data can also be accessed through the command line by subscribing to the IMU ROS topic. The raw IMU data is shown in **Figure 2.12**.

```
deepracer@amss-o93p:~/catkin_ws$ rostopic echo /imu
header:
  seq: 8044
  stamp:
    secs: 1612377785
    nsecs:  19211053
  frame_id: "base_imu_link"
orientation:
  x: 0.127677791359
  y: 0.756506797833
  z: -0.0490531072468
  w: 0.63952297777
orientation_covariance: [0.0025, 0.0, 0.0, 0.0, 0.0025, 0.0, 0.0, 0.0, 0.0025]
angular_velocity:
  x: 0.03
  y: 0.0
  z: 0.01
angular_velocity_covariance: [0.02, 0.0, 0.0, 0.0, 0.02, 0.0, 0.0, 0.0, 0.02]
linear_acceleration:
  x: -9.48041015625
  y: 0.85725890625
  z: -1.74784289062
linear_acceleration_covariance: [0.04, 0.0, 0.0, 0.0, 0.04, 0.0, 0.0, 0.0, 0.04]
---
header:
  seq: 8045
  stamp:
    secs: 1612377785
    nsecs:  39654016
  frame_id: "base_imu_link"
orientation:
  x: 0.12829715278
  y: 0.75665967445
  z: -0.0492787205578
  w: 0.639200739478
orientation_covariance: [0.0025, 0.0, 0.0, 0.0, 0.0025, 0.0, 0.0, 0.0, 0.0025]
angular_velocity:
  x: 0.07
  y: -0.0
  z: 0.04
angular_velocity_covariance: [0.02, 0.0, 0.0, 0.0, 0.02, 0.0, 0.0, 0.0, 0.02]
linear_acceleration:
  x: -9.54284679687
  y: 0.7277890625
  z: -1.88650585937
linear_acceleration_covariance: [0.04, 0.0, 0.0, 0.0, 0.04, 0.0, 0.0, 0.0, 0.04]
---
```

Figure 2.12: Raw IMU Data.

After installing the IMU, we began working on the crucial tasks of mapping and localization with the physical car. Before we could implement any control or RL algorithms on the physical car, the car must be able to create a map of its environment and be able to localize itself within the environment (identify its approximate position on the map it has created). Simultaneous localization and mapping (SLAM) algorithms allow us to use camera, LIDAR, and IMU data to create a map of the environment and localize the robot within the environment with respect to the map that it creates. The first step was to try SLAM with only one data source (either camera, LI-

DAR, or IMU), and later move on to fusing the data sources together. IMU raw data is very noisy and would require a lot of filtering, so we have decided to keep IMU data as a last resort. Camera data is very rich but it can only gather data from one direction (the direction the camera is facing), so it is inefficient to use as the sole source of data for SLAM. This leaves the LIDAR sensor, which is constantly rotating in a circle atop the car. There also exist many well know SLAM algorithms and implementations for LIDAR data, so we are currently working on LIDAR based SLAM.

The current SLAM algorithm we are using is called GMapping, which is combined with a ROS pose estimation utility called Laser Scan Matcher. Together, GMapping and Laser Scan Matcher allow for mapping and pose estimation within the environment. The mapping performance is shown in **Figures 2.13 and 2.14**.

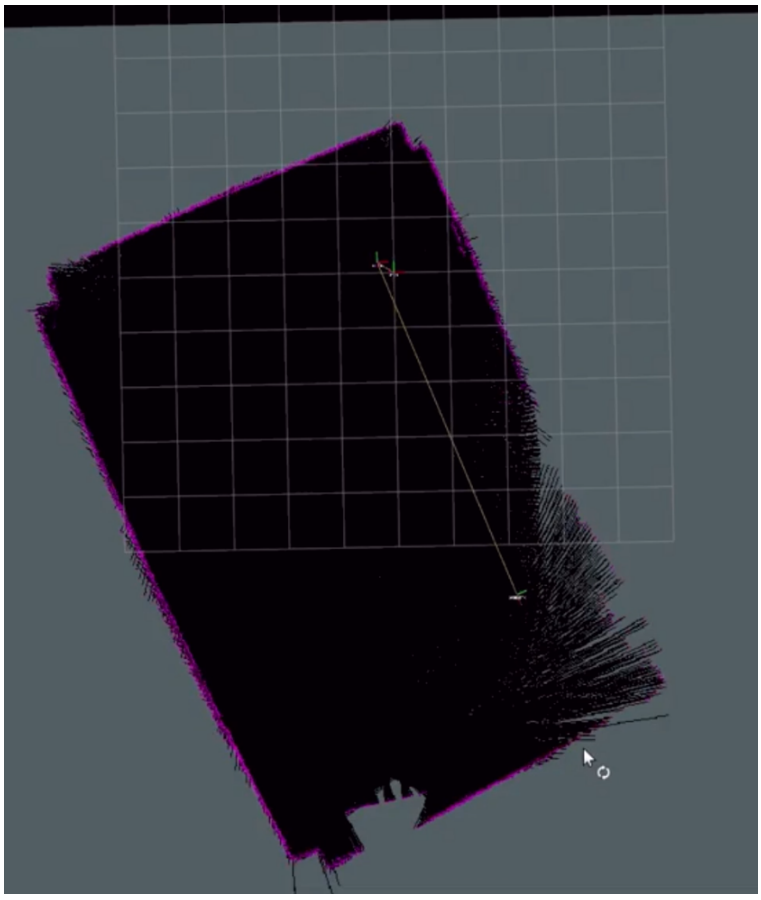

Figure 2.13: Mapping in Empty Lab.

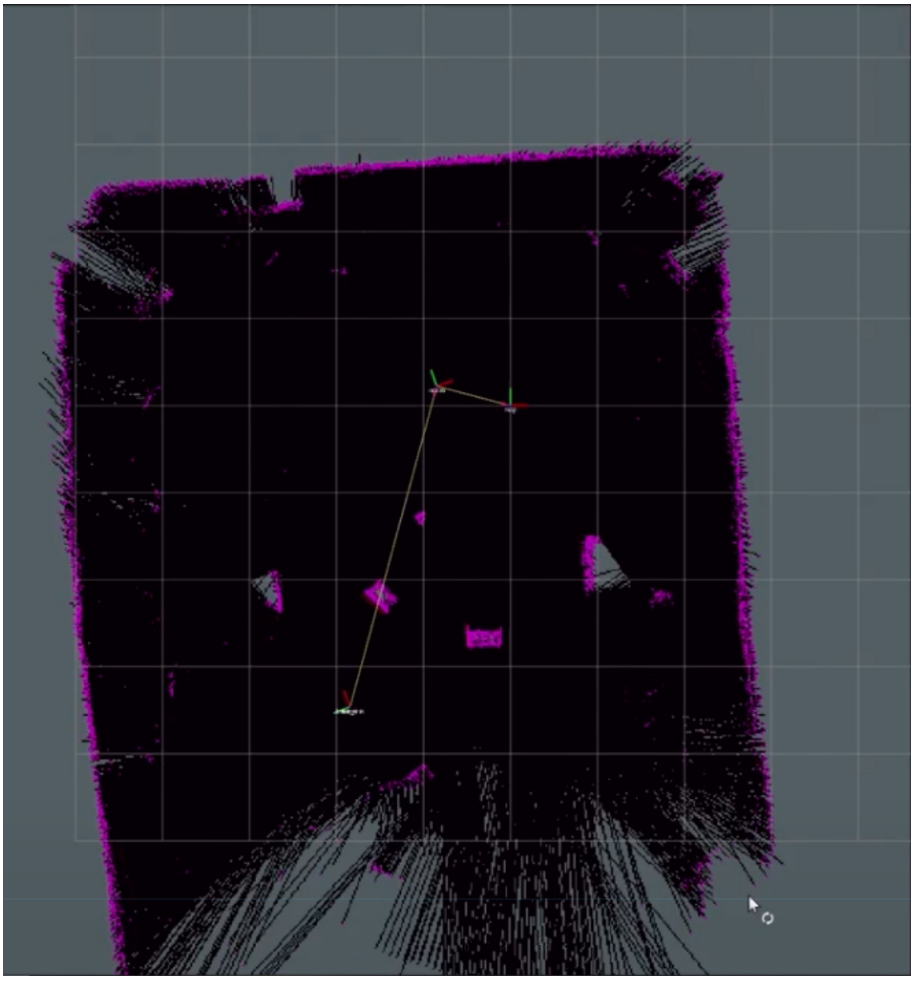

Figure 2.14: Mapping in Lab with Obstacles.

Figure 2.15 shows the results of localization validation testing. In this test, the physical DeepRacer car was driven in two ovals and then a figure 8 pattern within the ovals. A standard test for localization is to drive the car in a circle (or more generally, a closed loop), ensuring the car returns to exact same point from which it started. If the localization data/path reflects that the car has returned to its initial point, it can be said that the localization is working accurately. In order for the localization to work accurately, the car should be traveling at speeds under 0.5 m/s. Traveling faster that this significantly reduces the accuracy of the Laser Scan Matcher.

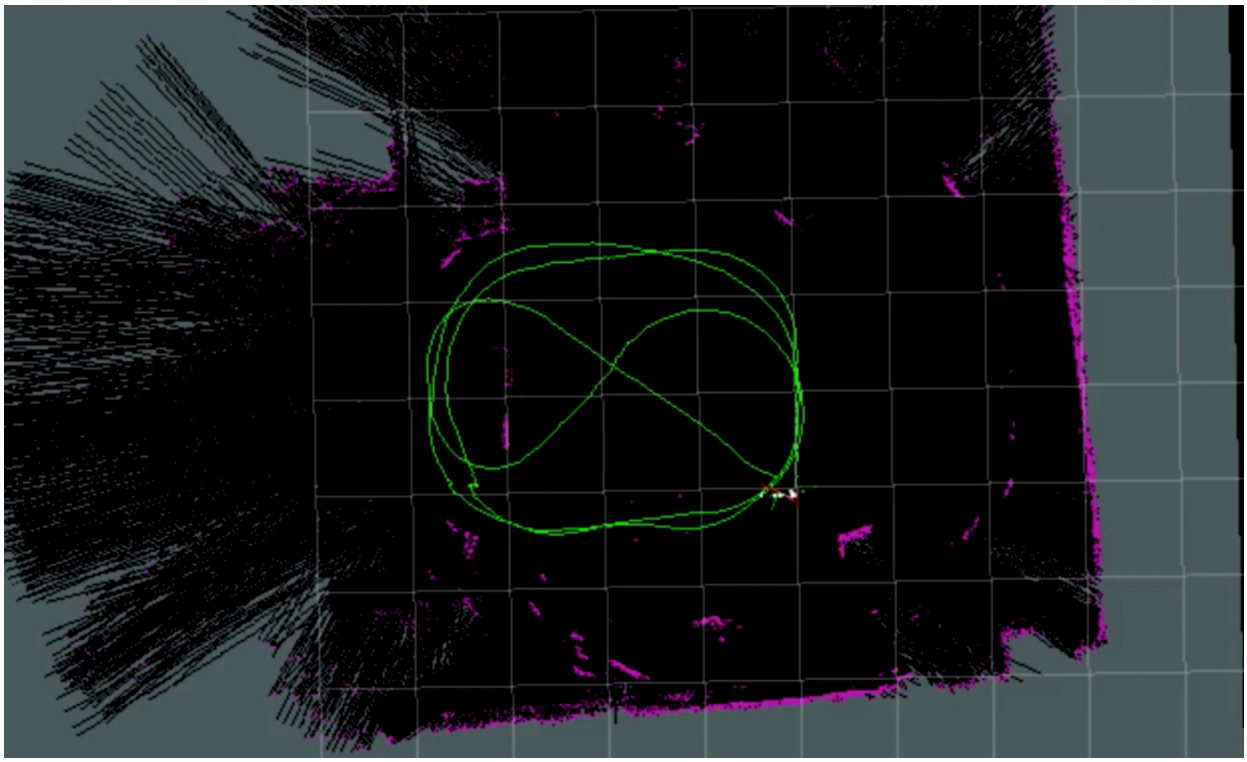

Figure 2.15: Localization

With the mapping and localization tasks complete, the next step was to deploy a basic PID controller in the physical DeepRacer for point to point navigation. This task step would also give us the first insights into the differences between simulation and reality. Since the car has two controls (throttle and steering), two PID controllers would have to be implemented (requires gain tuning, 3 coefficients per controller), one for each control input. To start off simple, a straight line controller was designed (goal point would straight in front of the start point, so no steering input would be required). Waypoints approximately 1m apart on a straight line were given to the PID controller and the resulting path is show in **Figure 2.16**. The curvature of the path is due to a steering offset present in the physical car, and can be corrected for. The performance of this controller was decent; while it was able to navigate to the 1m apart waypoints with fair accuracy, the localization became less and less accurate as the car was running, which in turn caused the navigation to become less accurate. This drift in localization accuracy was likely caused by "jerky"

(or unsmooth) motion of the car. Also, while accelerating, the car built up momentum, so even though it had turned off the throttle by the time it reached its goal point (meaning the controller was working as it should), the momentum often carried the car beyond the goal point. After including the PID Controller for steering, the results were not very good, as shown in **Figure 2.17**. With the steering PID included, the car was not able to navigate well beyond the first waypoint; it lost track of its localization and therefore went out of control. PID tuning is already quite a tedious and laborious task, especially for a very sensitive control input like steering, and keeping extra factors like momentum and steering offset in account only complicated the process further. The lack of success achieved by PID controllers for this task clearly highlight the benefit of RL based controllers, which are trained in a simulator and do not require crude trial and error gain tuning like PID controllers. This is because in RL, the agent will learn the optimal control, whereas in PID there is no notion of the agent learning on its own, which is what makes gain tuning in PID a tedious task for the designer. Therefore, instead of spending more time on PID, the next step was to explore RL.

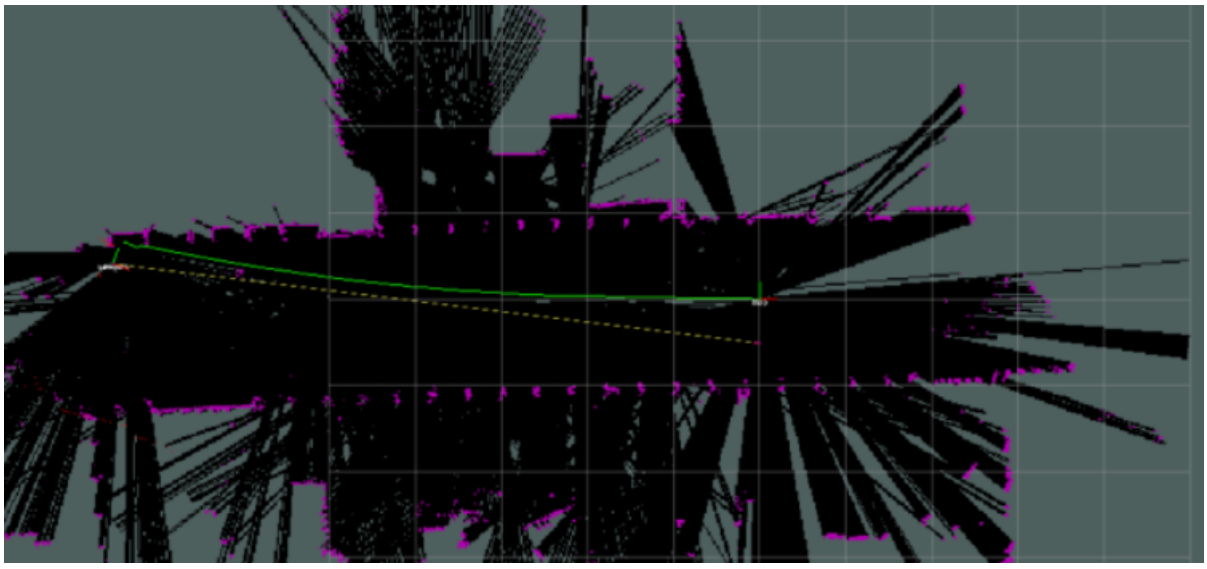

Figure 2.16: PID Straight Line Path

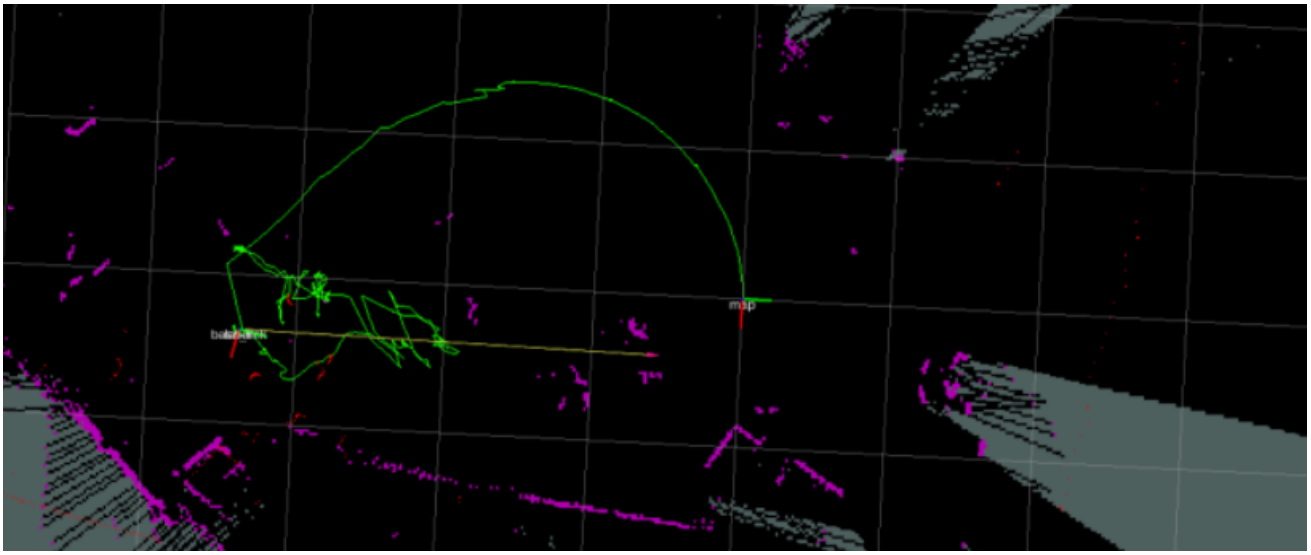

Figure 2.17: PID Throttle and Steer Path

2.4 RL Simulations

The RL training was done in an office environment in Gazebo, shown in **Figure 2.18**. This environment setting was chosen because it can facilitate a wide variety of training scenarios. For example, large, open settings can be simulated in the large office rooms, and small, crowded settings can be simulated in the narrow hallways and small office rooms.

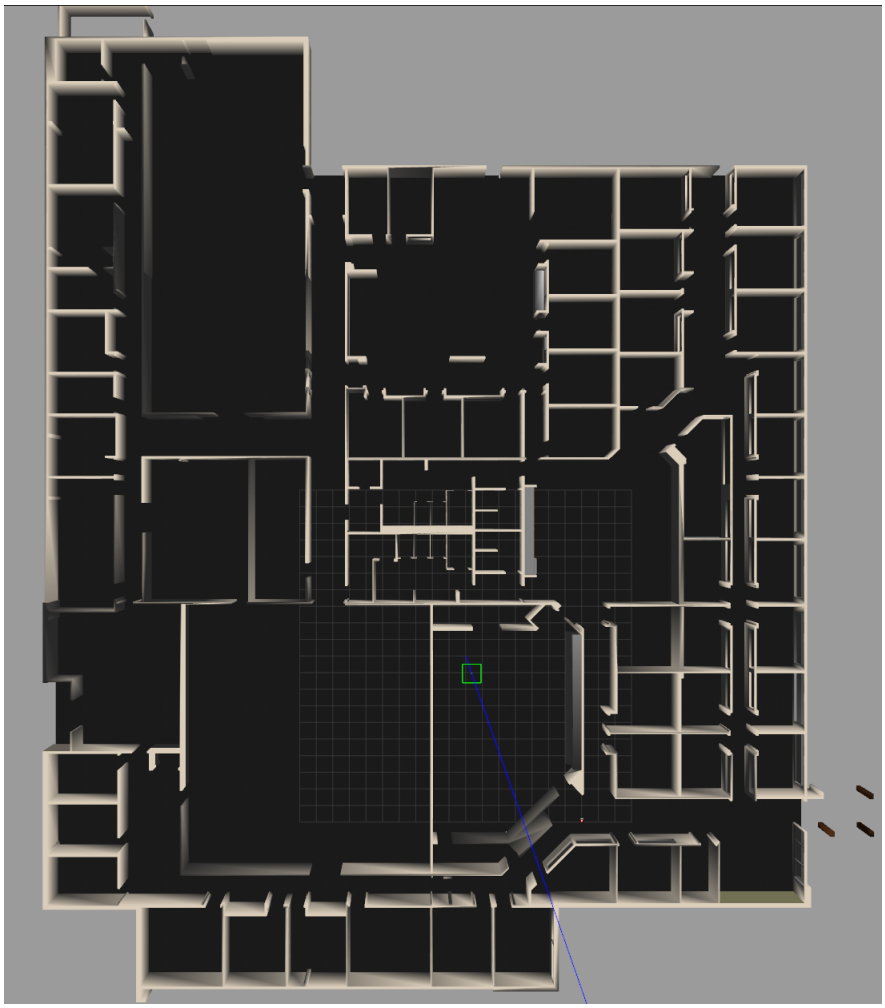

Figure 2.18: Office Environment in Gazebo

The first round of training was set up with a fixed initial starting point (start state was the coordinate (-10, -7)) and fixed goal point (goal state was the point (-8, -8)). The reason for choosing start/end points that area relatively close to each other for training is because this is exactly what the agent would be expected to do in a real autonomous navigation problem. Any long distance path can be decomposed in to a set of close by waypoints, which would successively be given to a trained agent for it to follow. The Soft Actor-Critic (SAC) algorithm, which has been successful in many RL applications over in the past few years, was used to train the agent. The training was set up for 200,000 timesteps, with a maximum episode length of 300 timesteps and a Replay Buffer

Size of 50,000. Throughout the duration of the training, 1250 episodes were completed, of which 845 resulted in the agent successfully reaching the goal state. The total time taken for this training was around 10 hours. The training plots are shown in **Figures 2.19, 2.20, and 2.21**.

Figure 2.19: Training Reward

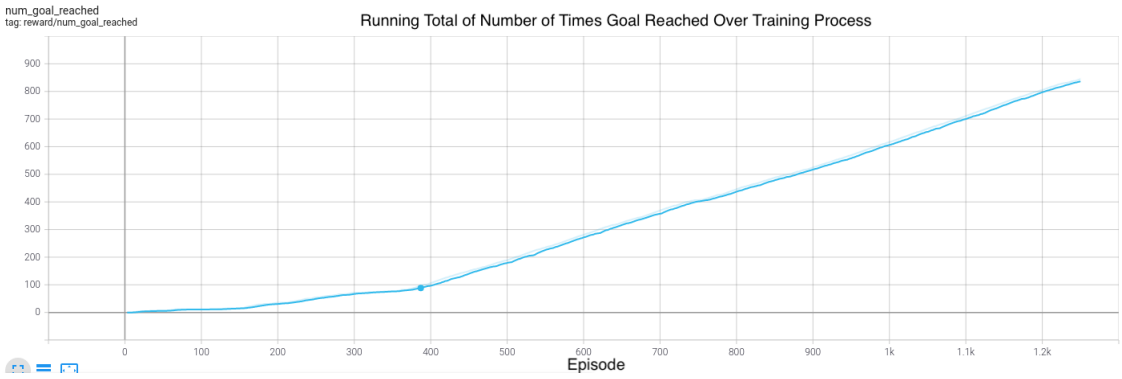

Figure 2.20: Training Goals Reached

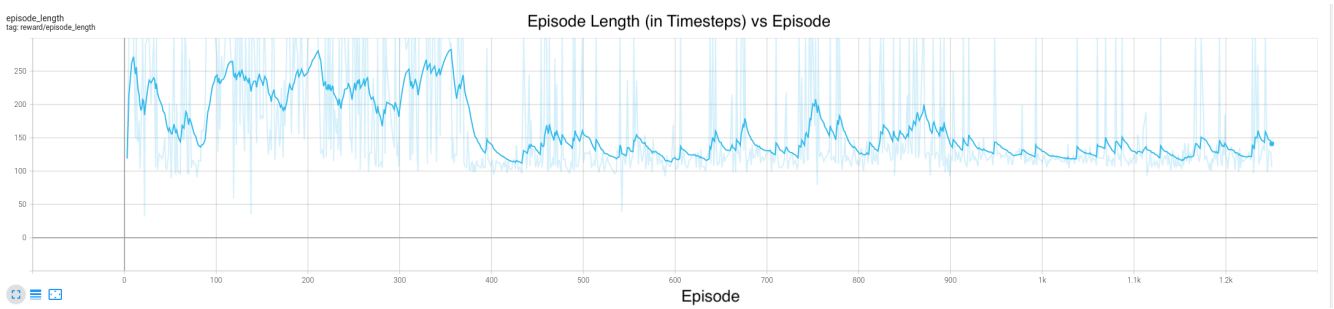

Figure 2.21: Training Episode Length

Figure 2.19 shows the reward the agent received in each episode throughout the training process. Clearly, as time went on, the agent was learning the proper way to ask, as reflected by the reward increasing. The reward converged to around 160 around halfway through the training. Furthermore, **Figure 2.20** shows that after episode 400, the agent was reaching the goal consistently in each episode, meaning it had learned how to act around episode 400, meaning the training could have been stopped earlier. Furthermore, **Figure 2.21** shows that when the agent learned the optimal policy (around episode 400), the timesteps per episodes drastically reduced as it wasted less time taking wrong actions and instead directly took the right actions. To confirm that the the model was indeed trained, the model was evaluated (given start point (-10, -7) and goal point (-8, -8)) and its trajectory was plotted in **Figure 2.22**, which clearly show that the agent takes a near optimal path from start to goal, and reaches the goal within the tolerance distance of 0.2.

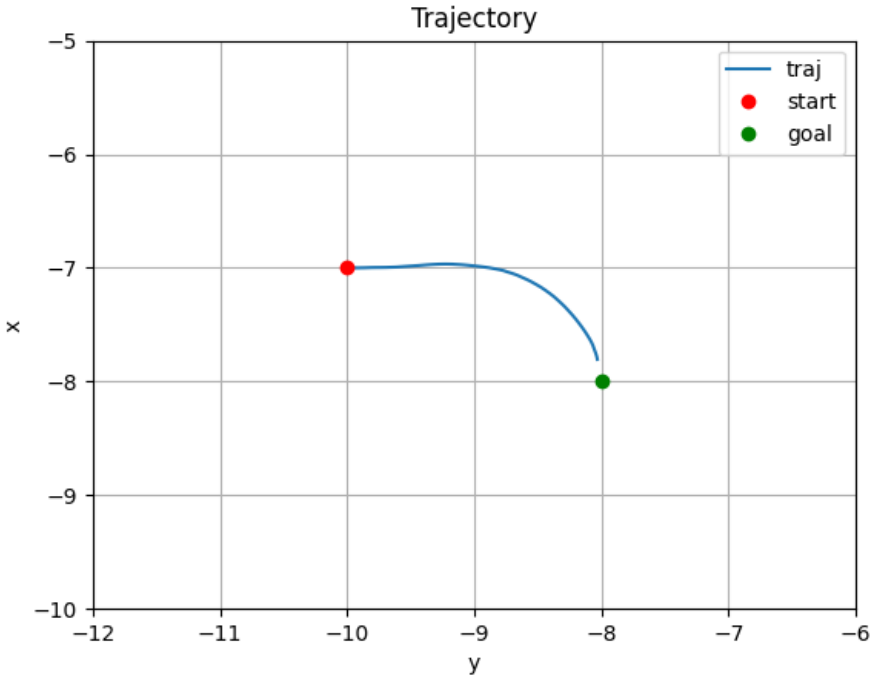

Figure 2.22: Evaluation Trajectory

While the evaluated model works well for the start/end points it was trained for, when different start and/or end points are given, the agent fails to reach the given end point. This indicates that with the current training setup, models that are trained are not robust. To make them more robust, the next step will be to randomize start and end points in the training, instead of just having a fixed start and end point. While this will surely increase training time, the agent should learn to to navigate to arbitrary points from arbitrary start locations.

3. CONCLUSION

The goal of this project was to explore reinforcement learning for autonomous vehicles, and hopefully establish RL as a viable alternative to traditional control methods for autonomous navigation and driving related tasks. In the 8 months that this project has been going on, a lot of progress has been made. When this project started, it was merely an idea of trying to deploy an RL algorithm on a robotic car. This seemingly straightforward task grew into a large learning experience, which showed me the stages of an autonomous vehicle project from start to finish. We started with path planning, and came up with a way to modify the popular A* algorithm to be suitable for autonomous driving purposes. The next step was to work with control algorithms that would be able to actuate the car to follow the generated path. Here we explored PID control and RL for actuating a vehicle in simulation and reality. In simulation, it was a tedious process to tune a PID controller to follow a path, which highlighted the need for a learning-based method like RL. To explore RL in simulation, many efforts went into getting a simulation environment where an agent could interact with the environment and use its experiences to learn an optimal policy with the aid of an RL algorithm. With the simulator up and running, an RL model was successfully trained for a waypoint navigation task. The trained model, while accurate and consistent, lacked robustness; it was only good to navigate between the two points it was trained for. The next steps in this would be to come up with a training scheme which trains for navigation between certain random waypoints to make the model more robust.

To deploy algorithms on the physical car, first the challenge of mapping and localization had to be solved, since a map of the physical environment cannot be assumed as known, and the car is not equipped with GPS functionality. After a lot of testing, and with the help of the LIDAR sensor and some ROS pacakges, the mapping and localization problem was solved and the car was ready to test PID controllers and RL models in reality. Deployment of the PID controllers in reality proved to be largely unsuccessful. While a straight-line PID controller worked well, anything more

complex than that would require an extensive amount of gain tuning, thus once again highlighting the need for RL, a learning-based approach. At this point in time, the car is unable to run RL models because the ROS utilities used for mapping and localization are quite "heavy" on the car's computing ability. Solutions are currently being explored to run the heavy computation on a more powerful machine on the same network, and leave only the light computation to the physical car.

There is immense potential for future work on this project, as there are many directions that can be explored. Two areas I am particulary interested in are Mult-Agent RL and Hiearchical RL. Multi-agent RL [5] deals with RL for multiple agents (for example, a platoon of car or drones). Studying how multiple agents can learn/cooperate together is becoming increasingly important as we quickly move toward an autonomous future, where roads will be filled with many autonomous agents. Hierarchical RL [6] deals with breaking down larger, broad RL problems into smaller "sub-problems" [7]. This can be very useful because, instead of training for small, specific problem, through HRL an agent can be trained for many different tasks on a broader RL problem. Regarding the implementation on the physical car, Sim2Real is an interesting direction to explore. While algorithms may work well in simulation, it is not necessary (or likely) that they will easily transfer from simulation to reality. There will be challenges (gaps) when adapting a simulated model for deployment in reality. For example, it is unlikely that a simulator can perfectly model an environment and all of its "governing laws", or there may be certain issues in reality (such as imperfect battery life or network delays) that do not even occur in simulation, leading to a gap between simulation and reality [8]. Identifying and bridging these gaps is a very exciting area which will only grow as RL becomes more prominent in our society.

Overall, the results from the RL training in this project look very promising, much more so than basic PID control. As the training setup is improved and advanced, we are confident that a well-trained, robust RL model can be produced for autonomous driving, and more focus can go into transferring this model to the physical car.

REFERENCES

[1] M. E. Abed, M. Aly, H. H. Ammar, and R. Shalaby, "Steering control for autonomous vehicles using pid control with gradient descent tuning and behavioral cloning," in *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES)*, pp. 583–587, 2020.

[2] M. Babu, Y. Oza, A. K. Singh, K. M. Krishna, and S. Medasani, "Model predictive control for autonomous driving based on time scaled collision cone," in *2018 European Control Conference (ECC)*, pp. 641–648, 2018.

[3] S. Bansal, V. Tolani, S. Gupta, J. Malik, and C. Tomlin, "Combining optimal control and learning for visual navigation in novel environments," 2019.

[4] B. Balaji, S. Mallya, S. Genc, S. Gupta, L. Dirac, V. Khare, G. Roy, T. Sun, Y. Tao, B. Townsend, E. Calleja, S. Muralidhara, and D. Karuppasamy, "Deepracer: Educational autonomous racing platform for experimentation with sim2real reinforcement learning," 2019.

[5] K. Zhang, Z. Yang, and T. Başar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," 2019.

[6] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman, "Meta learning shared hierarchies," 2017.

[7] B. Hengst, *Hierarchical Reinforcement Learning*, pp. 495–502. Boston, MA: Springer US, 2010.

[8] L. Weng, "Domain randomization for sim2real transfer," *lilianweng.github.io/lil-log*, 2019.